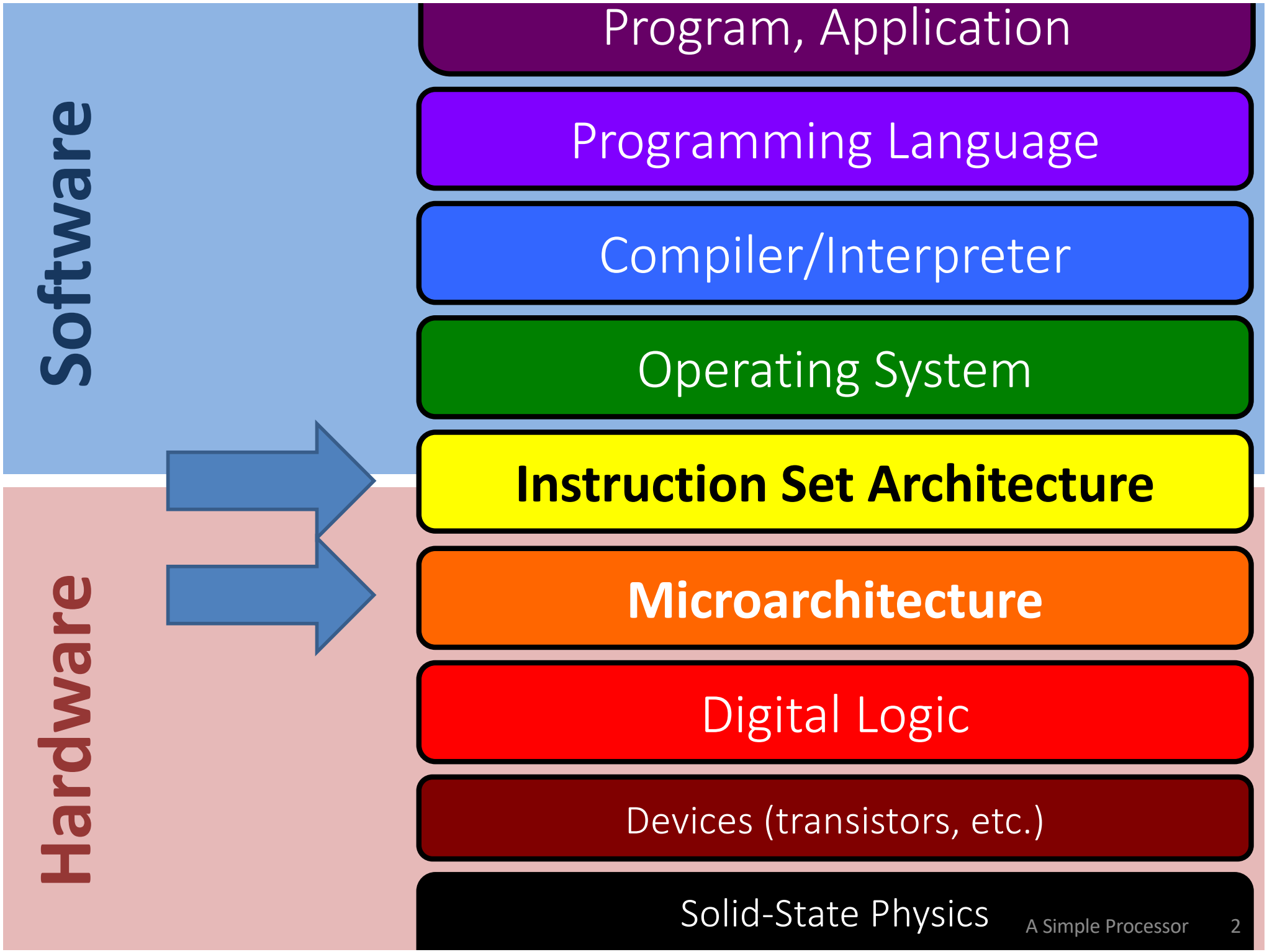
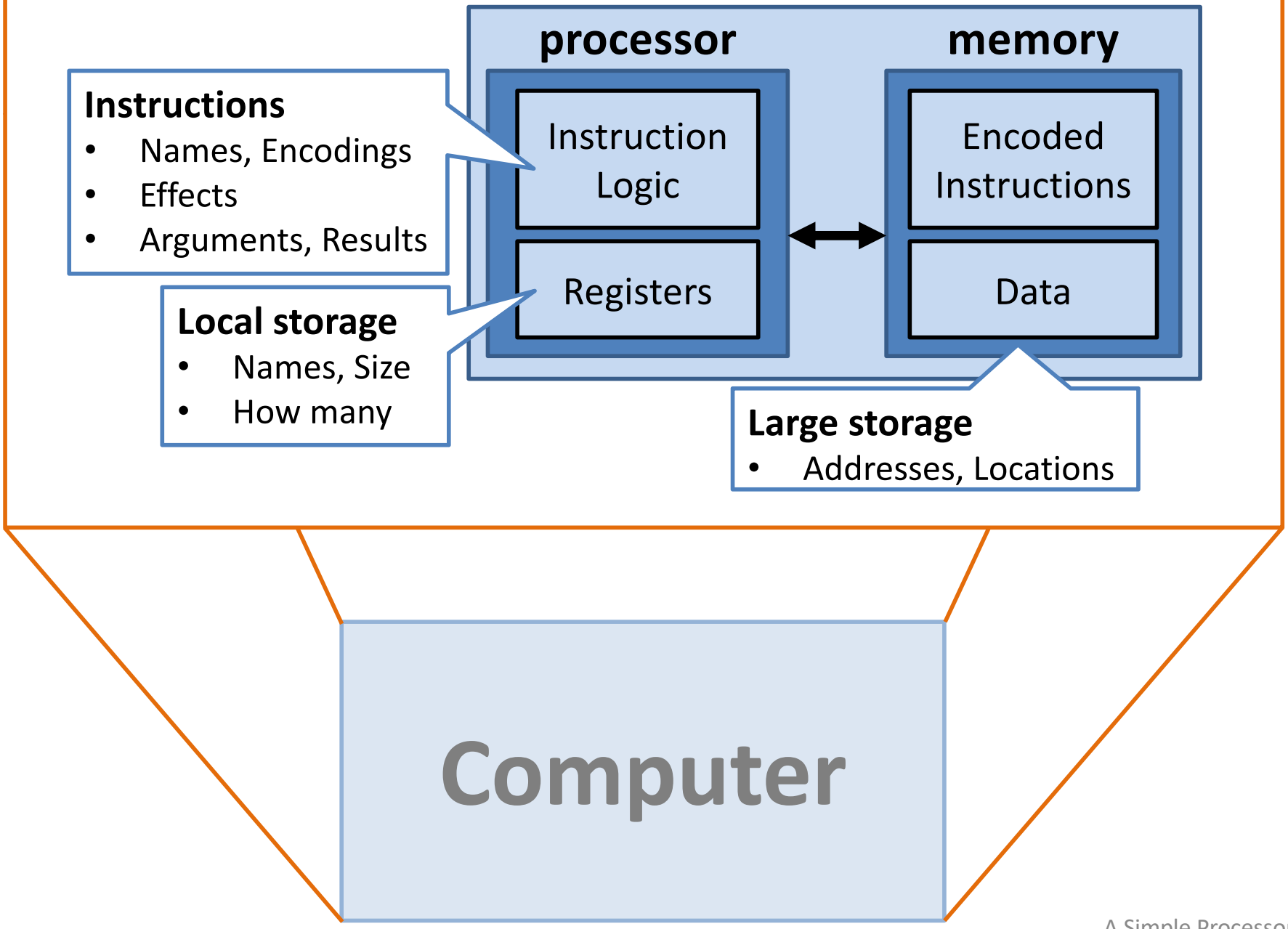
A Simple Processor

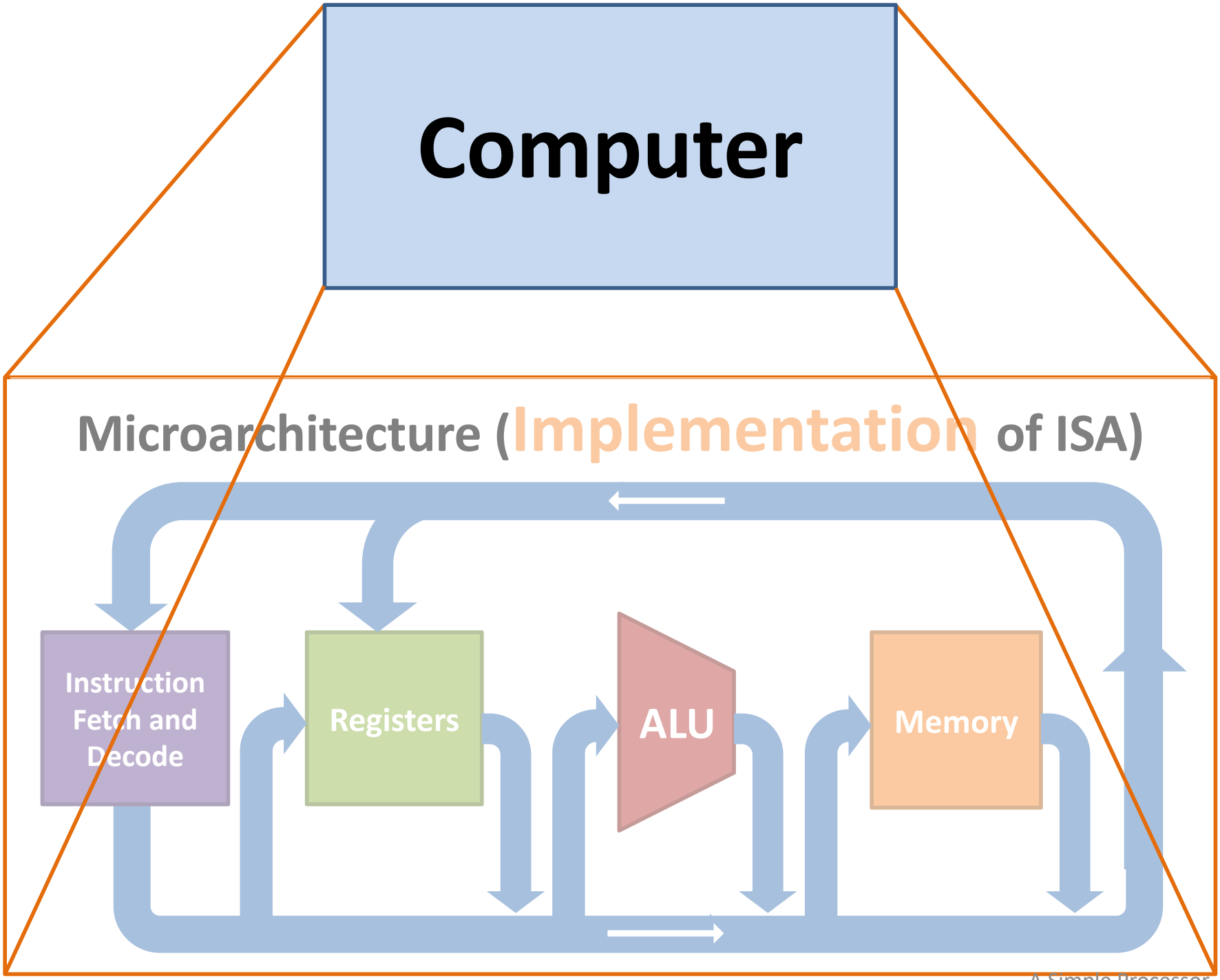
Contains solutions for slides 8–11

1. A simple Instruction Set Architecture
2. A simple microarchitecture (implementation): Data Path and Control Logic


Software
Hardware
Program, Application
Programming Language
Compiler/Interpreter
Operating System
Instruction Set Architecture
Microarchitecture
Digital Logic
Devices (transistors, etc.)
Solid-State Physics

Instruction Set Architecture (HW/SW Interface)




Computer
Microarchitecture (Implementation of ISA)
Instruction Fetch and Decode
Registers
ALU
Memory

HW ISA

An example made-up instruction set architecture

Word size = 16 bits

- Register size = 16 bits.
- ALU computes on 16-bit values.

Memory is byte-addressable, accesses full words (byte pairs).

16 registers: R0 - R15

- R0 always holds hardcoded 0
- R1 always holds hardcoded 1
- R2 – R15: general purpose

Instructions are 1 word in size.

Separate *instruction memory*.

Address	Contents
0	First instruction, low-order byte
1	First instruction, high-order byte
2	Second instruction, low-order byte
…	…

Program Counter (PC) register

- holds address of next instruction to execute.

R: Register File

Reg	Contents	Reg	Contents
R0	0x0000	R8	
R1	0x0001	R9	
R2		R10	
R3		R11	
R4		R12	
R5		R13	
R6		R14	
R7		R15	

M: Data Memory

Address	Contents	
0x0 – 0x1		
0x2 – 0x3		
0x4 – 0x5		
0x6 – 0x7		
0x8 – 0x9		
0xA – 0xB		
0xC – 0xD		
…		

Program Counter

PC

Processor Loop

1. *ins* ← IM[PC]
2. PC ← PC + 2
3. Do *ins*

HW ISA **Abstract Machine**

IM: Instruction Memory

Address	Contents
0x0 – 0x1	
0x2 – 0x3	
0x4 – 0x5	
0x6 – 0x7	
0x8 – 0x9	
…	

HW ISA Instructions

MSB **16-bit Encoding** LSB

Assembly Syntax	Meaning (R = register file, M = data memory)	Opcode	Rs	Rt	Rd
ADD R*s*, R*t*, R*d*	R[*d*] ← R[*s*] + R[*t*]	0010	*s*	*t*	*d*
SUB R*s*, R*t*, R*d*	R[*d*] ← R[*s*] - R[*t*]	0011	*s*	*t*	*d*
AND R*s*, R*t*, R*d*	R[*d*] ← R[*s*] & R[*t*]	0100	*s*	*t*	*d*
OR R*s*, R*t*, R*d*	R[*d*] ← R[*s*] | R[*t*]	0101	*s*	*t*	*d*
LW R*t*, *offset*(R*s*)	R[*t*] ← M[R[*s*] + *offset*]	0000	*s*	*t*	*offset*
SW R*t*, *offset*(R*s*)	M[R[*s*] + *offset*] ← R[*t*]	0001	*s*	*t*	*offset*
BEQ R*s*, R*t*, *offset*	If R[*s*] == R[*t*] then PC ← PC + 2 + *offset**2	0111	*s*	*t*	*offset*
JMP *offset*	PC ← *offset**2	1000	*o* *f*	*f* *s*	*e* *t*
HALT	Stops program execution	1111			

JMP offset is unsigned
All others are signed

Exercise #1 **Solutions**

R: Register File

Reg	Contents	Reg	Contents
R0	0x0000	R8	
R1	0x0001	R9	
R2		R10	
R3	**0xCAEB**	R11	
R4	**0x56BD**	R12	
R5	**0x42A9**	R13	
R6		R14	
R7		R15	

M: Data Memory

Address	Contents	
0x0 – 0x1	0xEB	0xCA
0x2 – 0x3	0xBD	0x56
0x4 – 0x5	**0xA9**	**0X42**
0x6 – 0x7		
0x8 – 0x9		
0xA – 0xB		
0xC – 0xD		
…		

Program Counter

PC

Processor Loop

1. *ins* ← IM[PC]
2. PC ← PC + 2
3. Do *ins*

HW ISA **Abstract Machine**

IM: Instruction Memory

Address	Contents
0x0 – 0x1	LW R3, 0(R0)
0x2 – 0x3	LW R4, 2(R0)
0x4 – 0x5	AND R3, R4, R5
0x6 – 0x7	SW R5, 4(R0)
0x8 – 0x9	HALT
0xA – 0xB	

Execution Table for *Ex. #1* (shows step-by-step execution)

Solutions

PC	Instr	State Changes
0x0	LW R3 0(R0)	R[3] ← M[R[0] + 0]] = M[0] = 0xCAEB; PC ← PC+2 = 0+2 = 2
0x2	LW R4, 2(R0)	R[4] ← M[R[0] + 2]] = M[2] = 0x56BD; PC ← PC+2 = 2+2 = 4
0x4	AND R3, R4, R5	R[5] ← R[*3*] & R[*4*] = 0xA942 ; PC ← PC+2 = 4+2 = 6
0x6	SW R5, 4(R0)	M[R[0] + 4] = M[4] ← R[5] = 0x42A9 ; PC ← PC+2 = 6+2 = 8
0x8	HALT	*Program execution stops*

The bytes are swapped from the Memory M picture on the previous page because it's assumed that the bytes are stored in so-called **Little Endian** order. E.g., for the byte pair 0xEB at address 0x0 and 0xCA at address 0x1, the byte at the lower address 0x0 is stored at the "little end" (LSB) of the 2-byte word. As we'll soon see, this is consistent with the byte ordering in the C programming language.

Exercise #2 **Solutions**

R: Register File

Reg	Contents	Reg	Contents
R0	0x0000	R8	~~0~~ ~~3~~ 6
R1	0x0001	R9	~~2~~ ~~1~~ 0
R2		R10	3
R3		R11	
R4		R12	
R5		R13	
R6		R14	
R7		R15	

M: Data Memory

Address	Contents	
0x0 – 0x1		
0x2 – 0x3		
0x4 – 0x5		
0x6 – 0x7		
0x8 – 0x9		
0xA – 0xB		
0xC – 0xD		
…		

Program Counter

PC

Processor Loop

1. *ins* ← IM[PC]
2. PC ← PC + 2
3. Do *ins*

HW ISA **Abstract Machine**

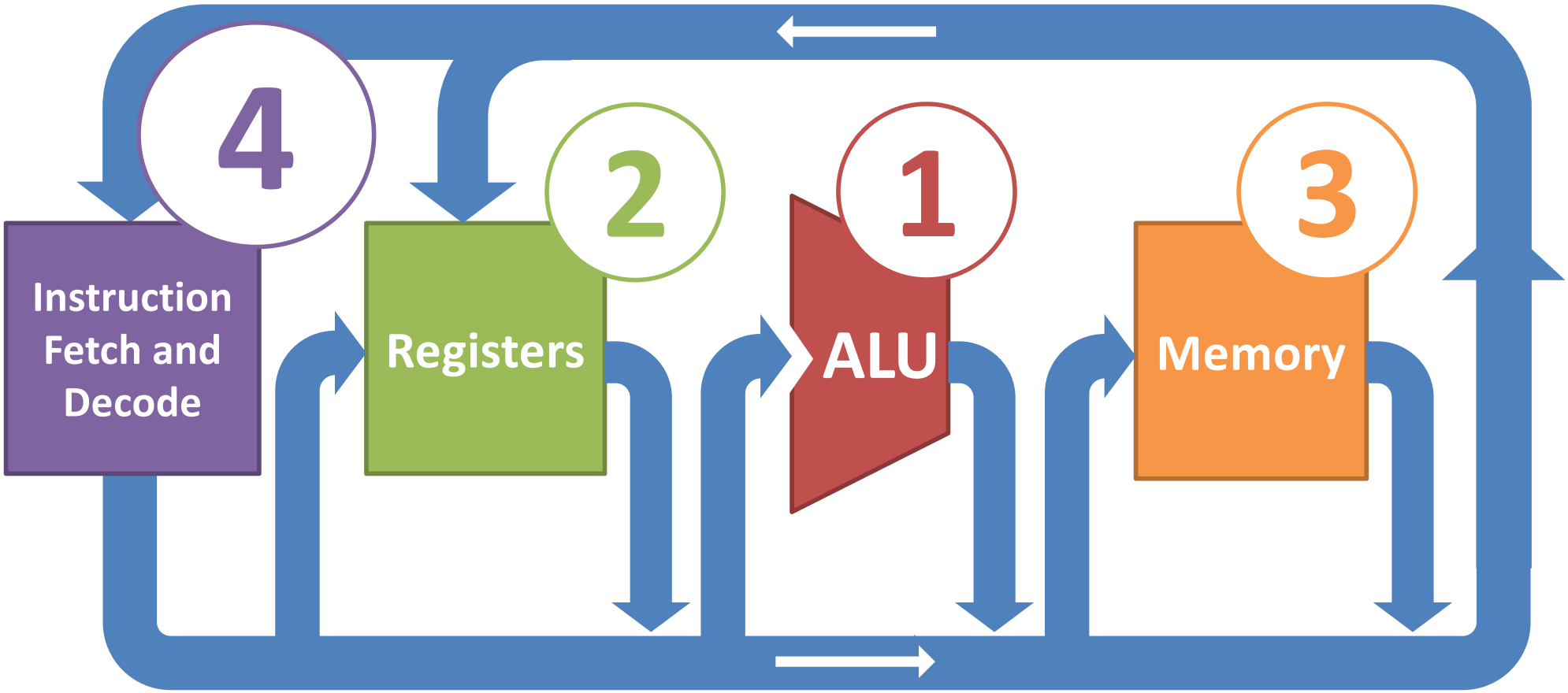
IM: Instruction Memory

Address	Contents
0x0 – 0x1	SUB R8, R8, R8
0x2 – 0x3	BEQ R9, R0, 3
0x4 – 0x5	ADD R10, R8, R8
0x6 – 0x7	SUB R9, R1, R9
0x8 – 0x9	JMP 1
0xA – 0xB	HALT

Execution Table for *Ex. #2* Solutions

PC	Instr	State Changes
0x0	SUB R8, R8, R8	R[8] ← R[8] – R[8] = 0; PC ← PC+2 = 0+2 = 2
0x2	BEQ R9, R0, 3	PC ← PC+2 = 2+2 = 4 (because 2 = R[9] ≠ R[0] = 0)
0x4	ADD R10, R8, R8	R[8] ← R[10] + R[8] = 3 + 0 = 3; PC ← PC+2 = 4+2 = 6
0x6	SUB R9, R1, R9	R[9] ← R[9] - R[1] = 2 – 1 = 1; PC ← PC+2 = 6+2 = 8
0x8	JMP 1	PC ← 2*1 = 2
0x2	BEQ R9, R0, 3	PC ← PC+2 = 2+2 = 4 (because 1 = R[9] ≠ R[0] = 0)
0x4	ADD R10, R8, R8	R[8] ← R[10] + R[8] = 3 + 3 = 6; PC ← PC+2 = 4+2 = 6
0x6	SUB R9, R1, R9	R[9] ← R[9] - R[1] = 1 – 1 = 0; PC ← PC+2 = 6+2 = 8
0x8	JMP 1	PC ← 2*1 = 2
0x2	BEQ R9, R0, 3	PC ← PC+2+(2*3) = 4+6 =10 (because 0 = R[9] = R[0] = 0)
0xA	HALT	*Program execution stops*

HW ARCH microarchitecture

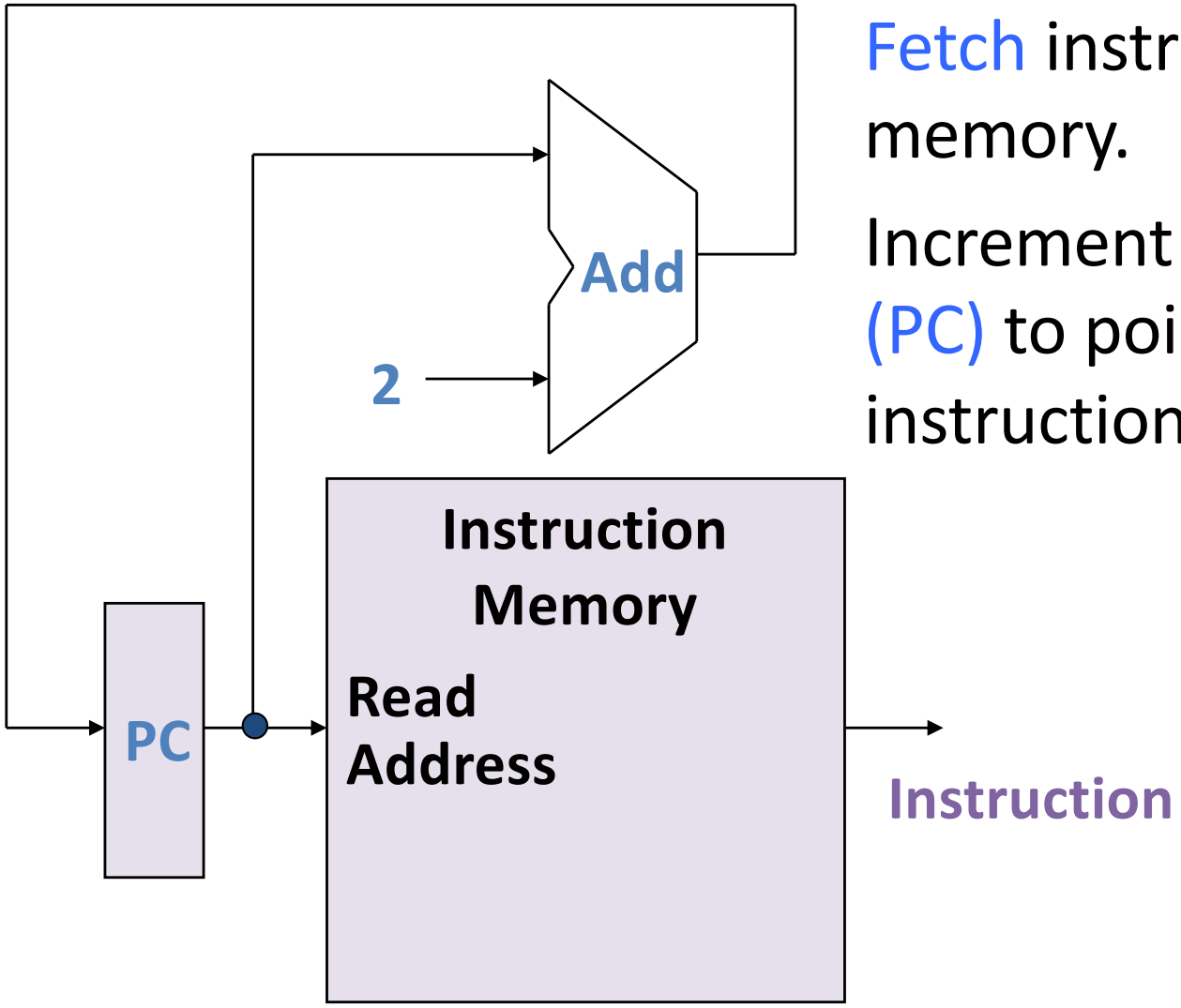


One possible hardware implementation of the HW ISA

Instruction Fetch
(default, unless branch or jump)

Processor Loop

1. *ins* ← IM[PC]
2. PC ← PC + 2
3. Do *ins*



Fetch instruction from memory.

Increment program counter (PC) to point to the next instruction.

Instruction Encoding: 3 formats

All have 4-bit opcode in MSBs

Arithmetic instructions:

- 2 source register IDs (Rs,Rt)
- 1 destination register ID (Rd)

15:12	11:8	7:4	3:0
opcode	*Rs*	*Rt*	*Rd*

Memory/branch instructions:

- address/source register ID (Rs)
- data/source register ID (Rt)
- 4-bit offset

15:12	11:8	7:4	3:0
opcode	*Rs*	*Rt*	*offset*

Jump instruction:

- 12-bit offset

15:12	11:0
opcode	*offset*

Arithmetic Instructions

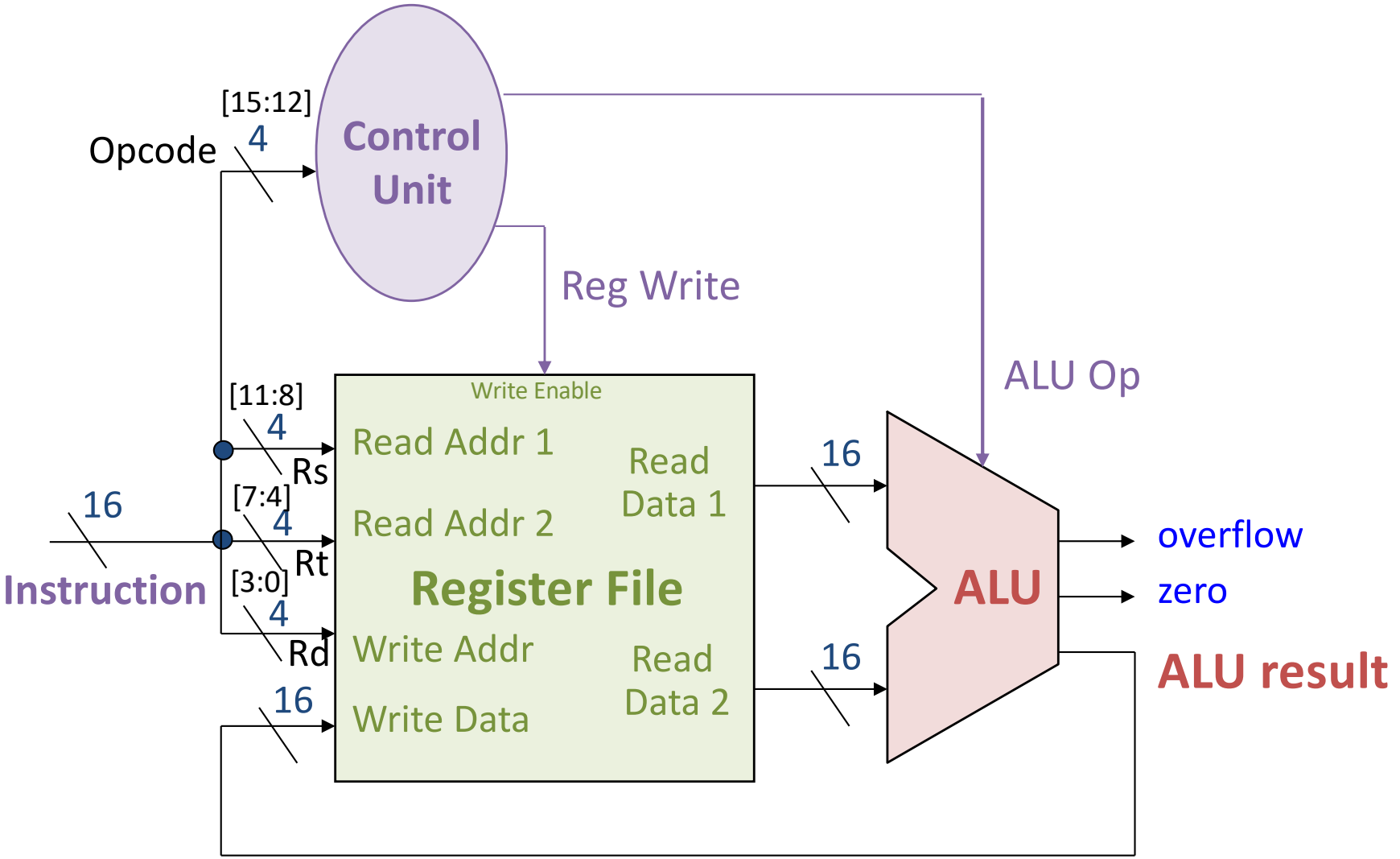
ADD R3, R6, R8

Opcode	Rs	Rt	Rd
0010	0011	0110	1000

16-bit Encoding

Instruction	Meaning	Opcode	Rs	Rt	Rd
ADD *Rs, Rt, Rd*	*R[d] ← R[s] + R[t]*	0010	*0-15*	*0-15*	*0-15*
SUB *Rs, Rt, Rd*	*R[d] ← R[s] – R[t]*	0011	*0-15*	*0-15*	*0-15*
AND *Rs, Rt, Rd*	*R[d] ← R[s] & R[t]*	0100	*0-15*	*0-15*	*0-15*
OR *Rs, Rt, Rd*	*Rd ← R[s] | R[t]*	0101	*0-15*	*0-15*	*0-15*
…					

Arithmetic Instructions:
Instruction Decode, Register Access, ALU



Memory Instructions

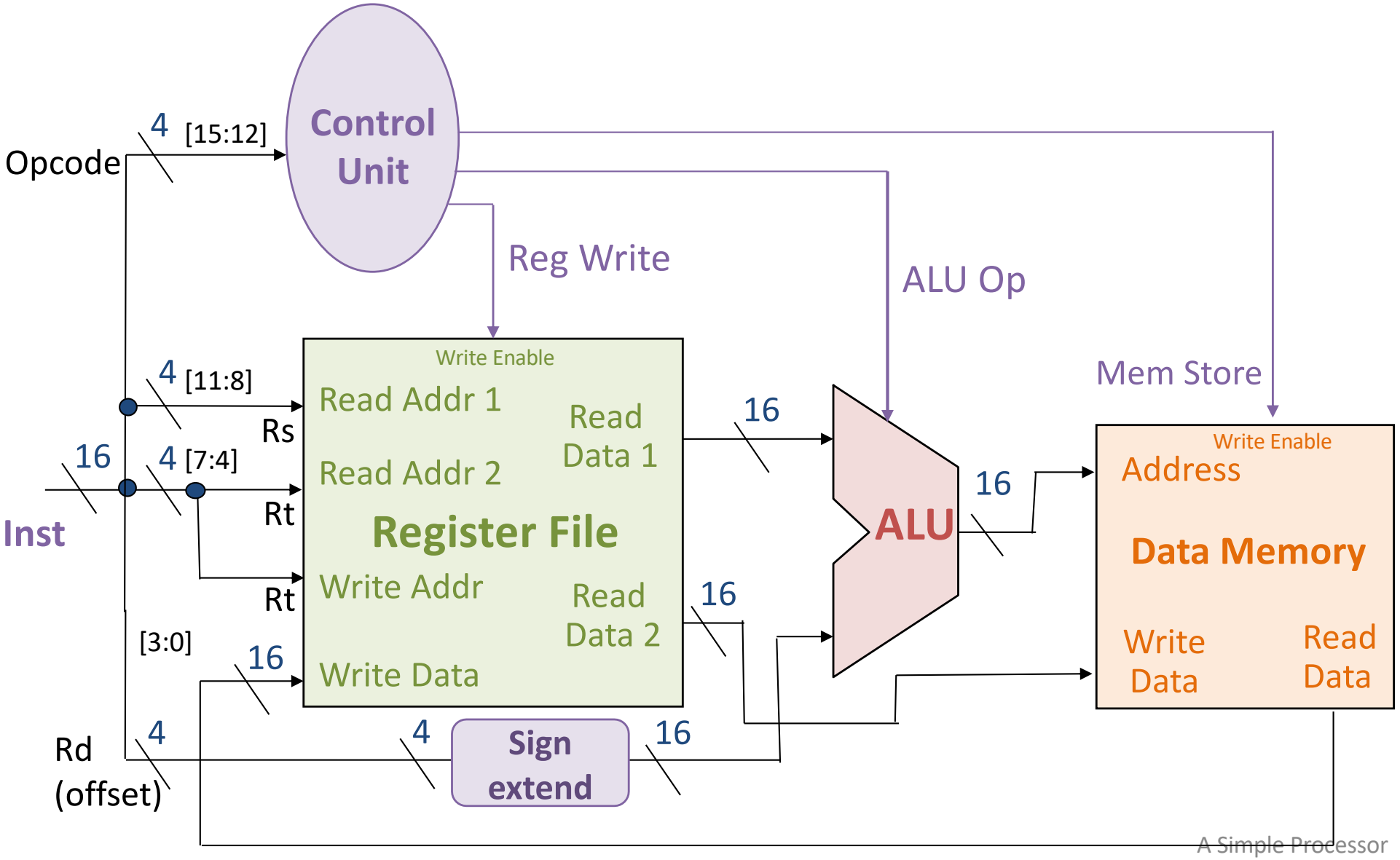
SW R6, -8(R3)

Opcode	Rs	Rt	Rd
0001	0011	0110	1000

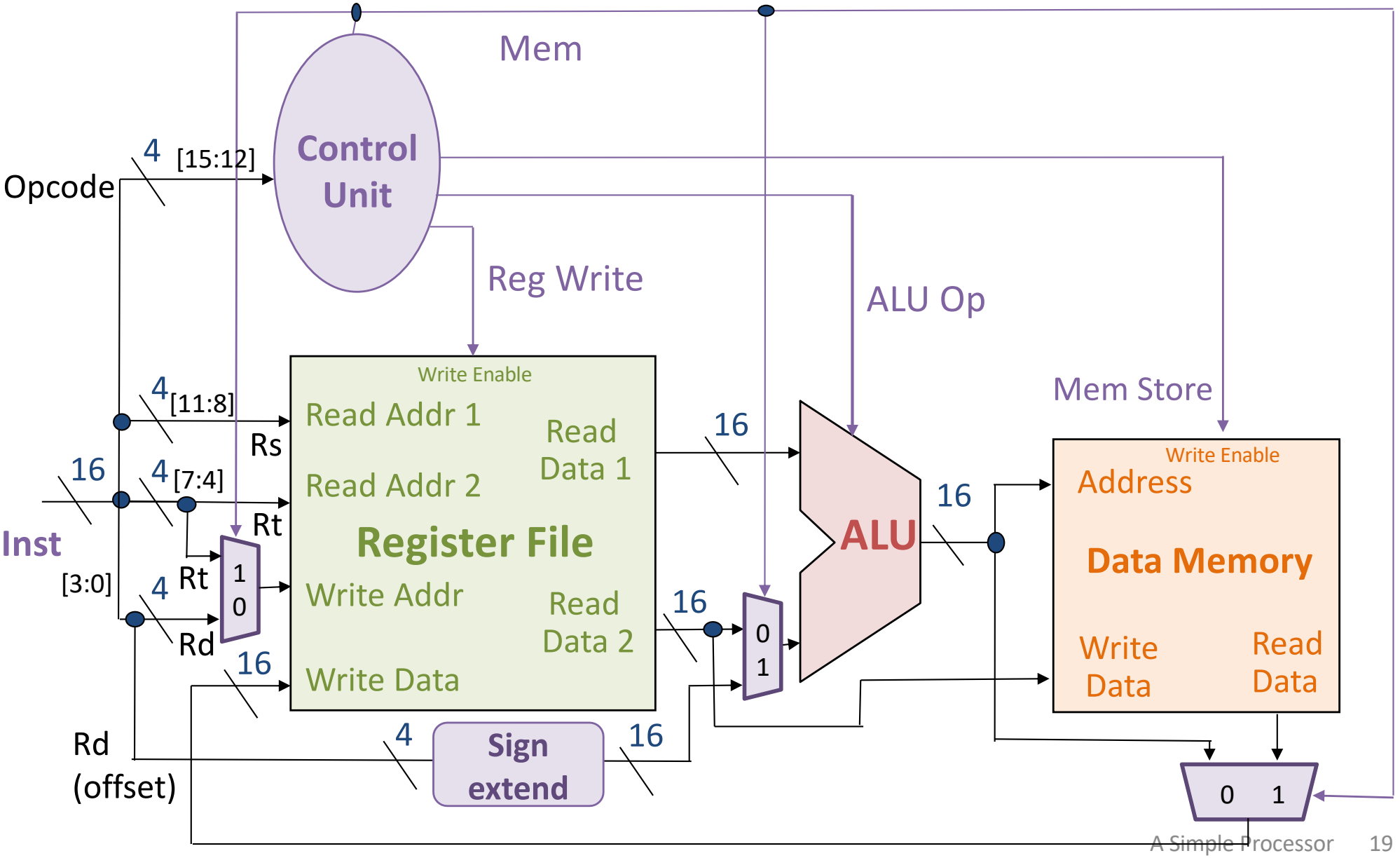
Instruction	Meaning	Op	Rs	Rt	Rd
LW *Rt, offset(Rs)*	*R[t] ← Mem[R[s] + offset]*	0000	*0-15*	*0-15*	*offset*
SW *Rt, offset(Rs)*	*Mem[R[s] + offset] ← R[t]*	0001	*0-15*	*0-15*	*offset*
…					

Memory Instructions:
Instruction Decode, Register/Memory Access, ALU

How can we support arithmetic **and** memory instructions? What's shared?



Choose between Arithmetic/Memory Instructions with MUXs



Control-flow Instructions

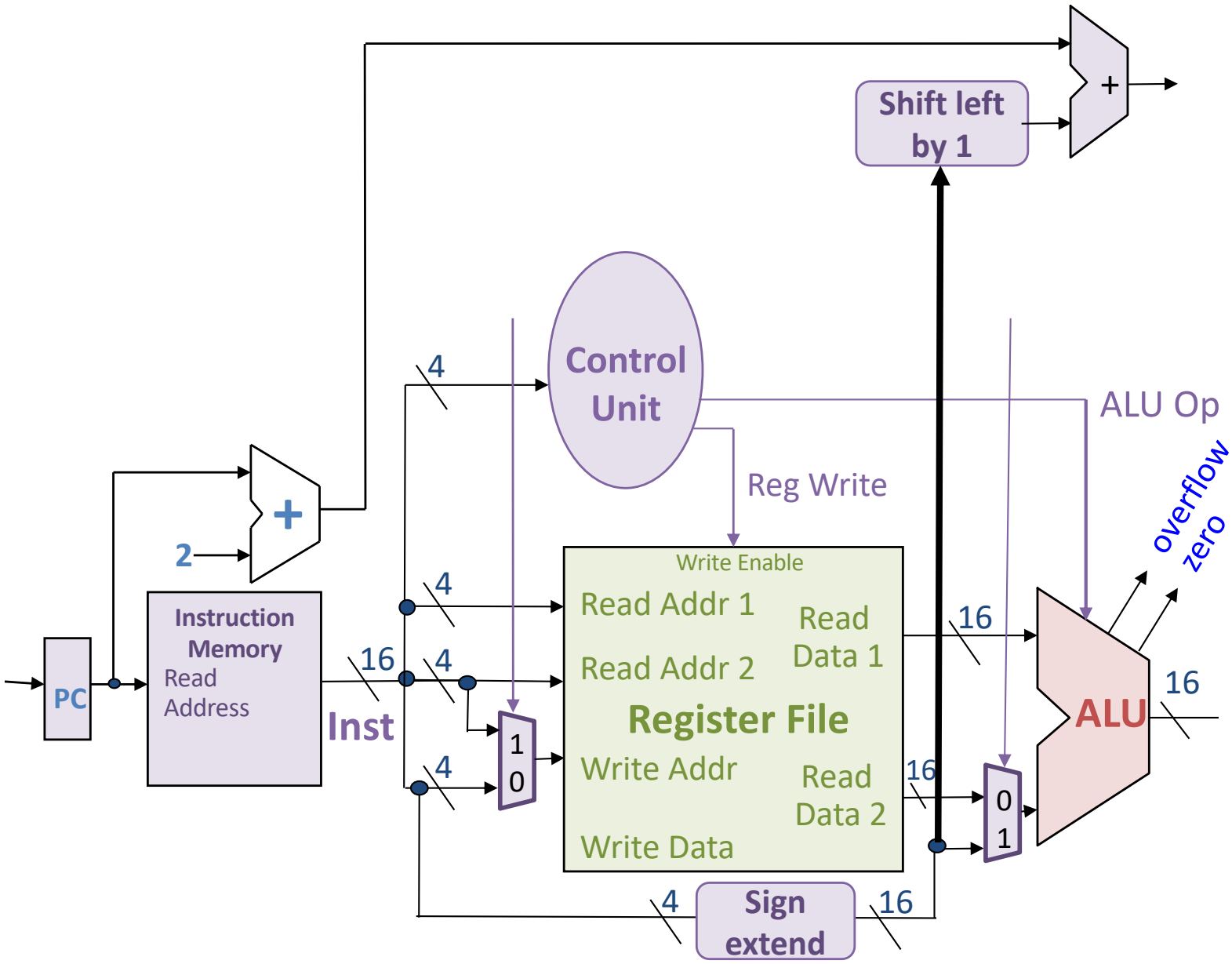
BEQ R1, R2, -2

Op	Rs	Rt	Rd
0111	0001	0010	1110

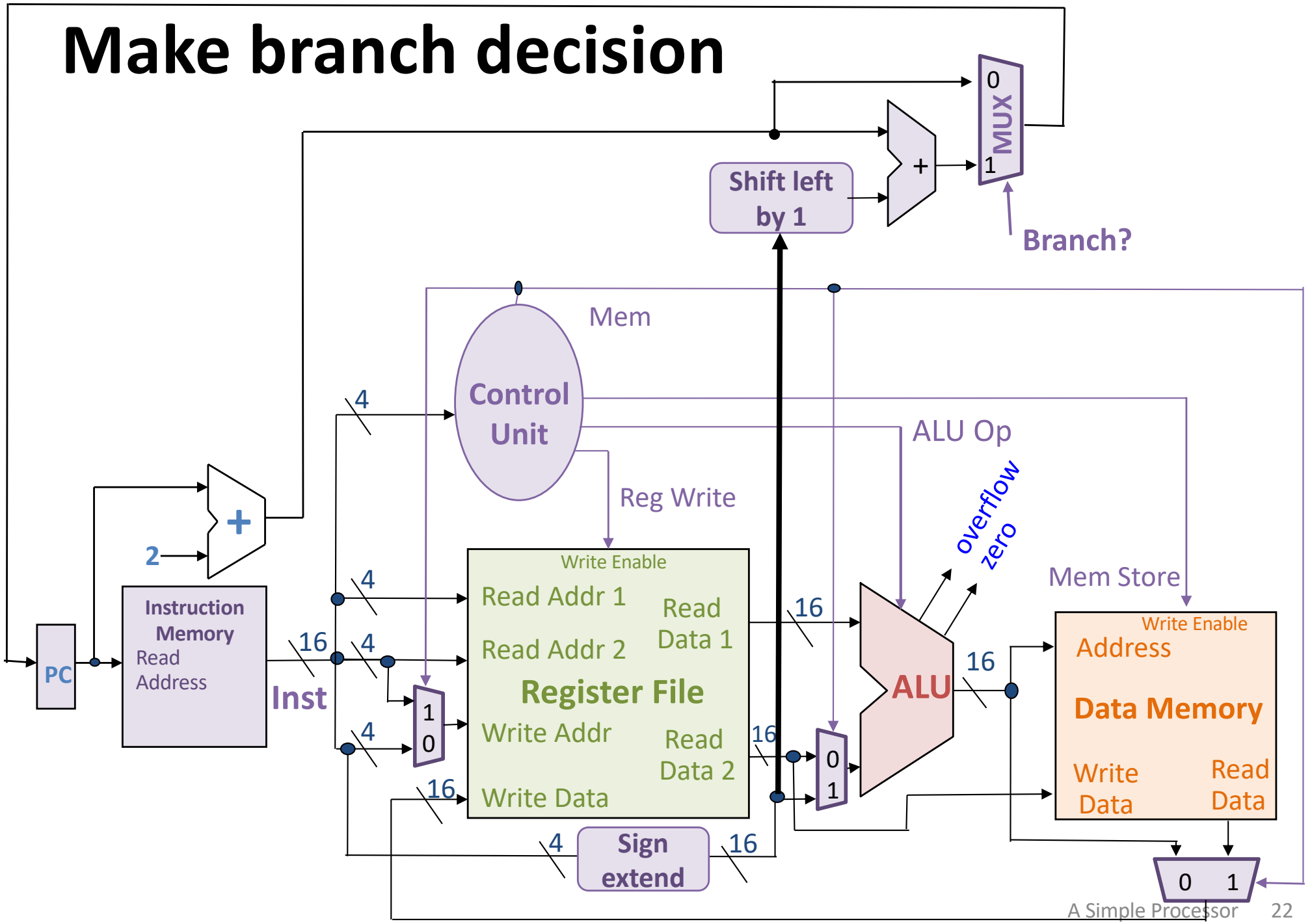
16-bit Encoding

Instruction	Meaning	Op	Rs	Rt	Rd
BEQ *Rs, Rt, offset*	*If R[s] == R[t] then* *PC ← PC + 2 + offset*2*	0111	*0-15*	*0-15*	*offset*
…					

Compute branch target for BEQ



Make branch decision



What's missing?

- Details of Control Unit
  - ALU op is **not** instruction opcode; some translation involved
  - Reg Write bit (for ADD, SUB, AND, OR, LW)
  - Mem Store bit (for SW)
  - Mem bit (arithmetic/memory MUX bit)
  - Branch bit (for BEQ)
- Implementation of JMP
- Implementation of HALT (basically stops the clock running the computer; we won't implement this)

See Lab 5 and Arch Assignment!

HW ARCH: not the only implementation

Single-cycle architecture

- Simple, "easily" fits on a slide (and in your head).
- One instruction takes one clock cycle.
- Slowest instruction determines minimum clock cycle.
- Inefficient.

Could it be better?

- Performance, energy, debugging, security, reconfigurability, …
- Pipelining
- OoO: out-of-order execution
- SIMD: single instruction multiple data
- Caching
- Microcode vs. direct hardware implementation
- … enormous, interesting design space of **Computer Architecture**